CS 321 Programming Languages

Intro to OCaml – Part I

Baris Aktemur

Özyeğin University

Last update made on Tuesday 24th October, 2017 at 09:18.

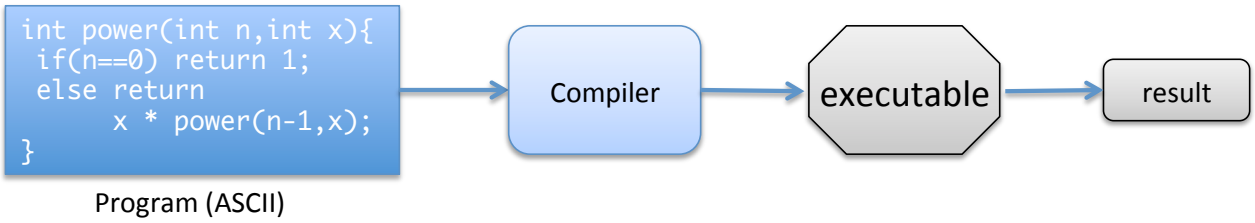
Much of the contents here are taken from Elsa Gunter and Sam Kamin's OCaml notes available at `http://courses.engr.illinois.edu/cs421`

Running a program

- **Compilation:** Convert a given program to a native (or native-like) format, e.g. object file, bytecode, etc., first. Then **execute** the native file.
- **Interpretation: Evaluate** a program directly, without a conversion to a native form.

Running a program

Compilation (e.g. C, C++):



Interpretation (e.g. OCaml, F#, Python):

OCaml

- OCaml programs can be both interpreted and compiled.
- We will use both models for executing programs.
- The interpreter is a so-called **REPL**, a read-eval-print-loop.
  - It **reads** what we type, **evaluates** our input, **prints** the results on the screen, then waits for the user's next input.
- **Evaluation** is the process of simplifying an expression as much as possible. An expression that cannot be simplified further is a **value**.
- By evaluation, we **reduce** programs to values.

```
[aktemur@ceviz]$ ocaml
        OCaml version 4.05.0
```

```
# 2 + 3;;
- : int = 5
# (* This is a comment *) 3 < 8;;
- : bool = true
# 3 = 2;; (* Use single '=' for equality *)
- : bool = false
```



```
# 2.5 + 5;; (* No implicit coercion *)

  2.5 + 5;;
  ------^

Error: This expression has type float but an expression was
         int
```

Declarations

With a declaration we "bind" a value to a name. The association of a name with a value is a "binding".

Declarations are made using the `let` keyword. After a declaration is made, the bound name can be used when declaring other names and in subsequent expressions.

Note:

Declarations can be made only at the top level.

Note:

I deliberately used the word "name", **not** "variable". This is because in OCaml, once bound, the value of a name cannot be changed.

Declarations

```
# let x = 2 + 3;; (* x is bound to 5 *)
val x : int = 5

# let test = 3 < 2;; (* test is bound to false *)
val test : bool = false

# x;;
- : int = 5
# x + 37;;
- : int = 42
# test && true;;
- : bool = false
```

'if'

```
# let a = 3;;
val a : int = 3

# let y = x + a + 5;;
val y : int = 13

(* 'if' is similar to (e1 ? e2 : e3) in C *)
# if y > a then 42 else 24;;
- : int = 42
# if not(y > a) then 42 else 24;;
- : int = 24
```


Environments

An **environment** is a set of bindings. It keeps record of what value is associated with a given name.

A key concept in programming language semantics and implementation.

Notation

- We will denote an environment as a table or a list of name-value associations.
- When a declaration is evaluated, we will append a new binding to the end of the table.
- When looking up the value of a name, we will search the table **from the end to the beginning**.

```
# let test = 3 < 2;;
val test : bool = false
```

Env: [test ↦ false]

```
# let a = 1
  and b = a + 4;;
val a : int = 1
val b : int = 5
```

Env: [b ↦ 5, a ↦ 1, test ↦ false]

```
# let a = 3;;
val a : int = 3
```

Env: [a ↦ 3, b ↦ 5, a ↦ 1, test ↦ false]

```
# a;;
- : int = 3
```

New bindings shadow the old!!!

Tuples

```
# let pair = (4, 6);;
val pair : int * int = (4, 6)
```

Env: [pair ↦ (4,6)]

```
# let s = (3, "hi", 4.5);;
val s : int * string * float = (3, "hi", 4.5)
```

Env: [s ↦ (3, “hi”, 4.5), pair ↦ (4,6)]

```
# let (a,b,c) = s;; (* (a,b,c) is a pattern *)
val a : int = 3
val b : string = "hi"
val c : float = 4.5
```

Env: [c ↦ 4.5, b ↦ “hi”, a ↦ 3, s ↦ (3, “hi”, 4.5), pair ↦ (4,6)]

```
# let a = 9 + 9;;
val a : int = 18
# let b = a < 10;;
val b : bool = false
```

Env: [b ↦ false, a ↦ 18, c ↦ 4.5, b ↦ “hi”, a ↦ 3, s ↦ (3, “hi”, 4.5), pair ↦ (4,6)]

Tuples

```
# (* Tuples can be nested *)
  let d = ((1,2,3), ("hi", 3.4), 'a');;

val d : (int * int * int) * (string * float) * char =
  ((1, 2, 3), ("hi", 3.4), 'a')

# let (p, (st, _), _) = d;; (* Patterns can be nested *)
                            (* _ matches everything *)
val st : string = "hi"
val p : int * int * int = (1, 2, 3)
```



```
# d;;
- : (int * int * int) * (string * float) * char =
  ((1, 2, 3), ("hi", 3.4), 'a')

# let (p, (st, _, f), _) = d;;

Error: This expression has type (int * int * int) * (string * float) * char
       but an expression was expected of type
         (int * int * int) * ('a * 'b * 'c) * 'd
       Type string * float is not compatible with type 'a * 'b * 'c
```

Exercise

Create a tuple for each of the given types.

```
# ???;;
- : int * (float * string) * float * char = ...


# ???;;
- : int * (float * string) * (float * char) = ...
```


Exercise

What is the environment at the end of the following OCaml session, assuming we start with an empty environment?

```
# let a = 5 + 7;;

# let b = 5 > 8;;

# let point = (5, 7, 9);;

# let a = 99;;

# let (a, b, c) = point;;

# let a = 55;;

# let point = (77, 88, 99);;

# let p = (a, b > 5, a + c);;
```

Java

Here is how you might define pairs in Java: Define a class `Pair`, and use its instances.

```
class Pair<A,B> {
  A first;
  B second;
  Pair(A first, B second) {
    this.first = first;
    this.second = second;
  }
}
...
new Pair<Integer, Pair<Float, String>>(42, new Pair<Float, String>(3.14, "hi"))
```

Argh... This is not as neat as OCaml.

Java

```
let (x, (y,z)) = p
```

would translate to

```
int x = p.first;
float y = p.second.first;
String z = p.second.second;
```

For triples, similar to `Pair`, define a class named `Triple`. But how about tuples of arbitrary size?

Functions

```
# let plusTwo = fun n -> n + 2;;
val plusTwo : int -> int = <fun>
```

Env: [plusTwo $\mapsto$ $\langle$n $\rightarrow$ n+2$\rangle$]

Functions are values as well. They go into the environment just like any other value, such as integer, string, tuple, etc.

```
# plusTwo 98;;
- : int = 100

# plusTwo;;
- : (int -> int) = <fun>
```


Functions

A shorter/cleaner syntax for

```
# let plusTwo = fun n -> n + 2;;
```

is

```
# let plusTwo n = n + 2;;
```

These two things are exactly the same.